Model-Based Test Selection for Infinite-State Reactive Systems

Bertrand Jeannet, Thierry Jéron, Vlad Rusu

Model-Based Test Selection for Infinite-State Reactive Systems*

Bertrand Jeannet[1], Thierry Jéron[2], and Vlad Rusu[2]

[1] INRIA Rhônes-Alpes, Montbonnot, France
Bertrand.Jeannet@inrialpes.fr
[2] IRISA/INRIA, Campus de Beaulieu, Rennes, France
{Thierry.Jeron,Vlad.Rusu}@irisa.fr

Abstract. This paper addresses the problem of off-line selection of test cases for testing the conformance of a black-box implementation with respect to a specification, in the context of reactive systems. Efficient solutions to this problem have been proposed in the context of finite-state models, based on the ioco conformance testing theory. An extension of these is proposed in the context of infinite-state specifications, modelled as automata extended with variables. One considers the selection of test cases according to test purposes describing abstract scenarios that one wants to test. The selection of program test cases then consists in syntactical transformations of the specification model, using approximate analyses.

1 Introduction and Motivation

Testing is the most used validation technique to assess the correctness of reactive systems. Among the aspects of software that can be tested, e.g., functionality, performance, timing, robustness, etc, the focus is here on conformance testing and specialized to reactive systems. This activity has been precisely described in the context of telecommunication protocols [15].

Conformance testing consists in checking that a black-box implementation of a system, only known by its interface and its interactions with the environment through this interface, behaves correctly with respect to its specification. Conformance testing then relies on experimenting the system with test cases, with the objective of detecting some faults with respect to the specification's external behaviour, or improve the confidence one may have in the implementation.

Despite the importance of the testing activity in terms of time and cost in the software life-cycle, testing practice most often remains ad hoc, costly and of rather poor quality, with severe consequences on the cost and quality of software. One solution to improve the situation is to automatize some parts of the testing activity, using models of software and formal methods. In this context, for more than a decade, *model-based testing* (see e.g., [6]) advocates the use of formal models and methods to formalize this validation activity. The formalization relies on models of specifications, implementations and test cases, a formal

* This paper is partly based on [27,18,28].

F.S. de Boer et al. (Eds.): FMCO 2006, LNCS 4709, pp. 47–69, 2007.

definition of conformance, or equivalently a fault model defining non-conformant implementations, test selection algorithms, and properties of generated test cases with respect to conformance.

In the context of reactive systems, the system is specified in a behavioral model which serves both as a basis for test generation, and as an oracle for the assignment of verdicts in test cases. Testing theories based on finite-state models such as automata associated to fault models (see e.g., the survey [20]), or labelled transition systems with conformance relations (see e.g., [29]) are now well understood.

Test generation/selection algorithms have been designed based on these theories, and tools like TorX [2], TGV [16], Gotcha [3] among others, have been developed and successfully used on industrial-size systems.

Despite these advances, some developments are still necessary to improve the automation of test generation. Crucial aspects such as compositionality (see e.g., [30]), distribution, real-time or hybrid behavior have to be taken into account for complex software. In this paper some recent advances made in our research group are reviewed, which cope with models of reactive systems with data.

In this paper, models of reactive systems called Input/Output Symbolic Transition Systems (ioSTS) are considered. These are automata extended with variables, with distinguished input and output actions, and corresponding to reactive programs without recursion. Their semantics can be defined in terms of infinite-state Input/Output Labelled Transition Systems (ioLTS). For ioLTS, the ioco testing theory [29] defines conformance as a partial inclusion of external behaviours (suspension traces) of the implementation in those of the specification. Several research works have considered this testing theory and propose test generation algorithms. The focus here is on off-line test selection, where a test case is built from a specification and a test purpose (representing abstract behaviours one wants to test), and further executed on the implementation. Test cases are built directly from the ioSTS model rather than from the enumerated ioLTS semantic model. This construction relies on syntactical transformations of the specification model, guided by an approximation of the set of states co-reachable from a set of final state.

2 Modelling Reactive Systems with Data Using ioSTS

The work presented in this paper targets reactive systems. A model inspired by I/O automata [24] is proposed and called ioSTS for *Input/Output Symbolic Transition Systems*. This model extends labelled transition systems with data, and is adequate as an intermediate model for imperative programs without recursion and communicating with their environment. The syntax is first presented, then its semantics in terms of transition systems, and the section finishes with the definition of the visible behavior of an ioSTS for testing.

2.1 Syntax of the ioSTS Model

An ioSTS is made of variables, input and output actions carrying communication parameters carried by actions, guards and assignments. As will be seen later, this model will serve for specifications, test cases and test purposes. One thus needs a model general enough for all these purposes.

One important feature of this model is the presence of variables, for which some notations need to be fixed. Given a variable v and a set of variables $V = \{v_1, \ldots, v_n\}$, $\mathcal{D}_v$ denotes the domain in which v takes its values, and $\mathcal{D}_V$ the product domain $\mathcal{D}_{v_1} \times \ldots \times \mathcal{D}_{v_n}$. An element of $\mathcal{D}_V$ is thus a vector of values for the variables in V. The notation $\mathcal{D}_{\boldsymbol{v}}$ is used for a vector $\boldsymbol{v}$ of variables. Depending on the context, a predicate $P(V)$ on a set of variables V may be considered either as a set $P \subseteq \mathcal{D}_V$, or as a logical formula, the semantics of which is a function $\mathcal{D}_V \rightarrow \{true, false\}$. An assignment for a variable v depending on the set of variables V is a function of type $\mathcal{D}_V \rightarrow \mathcal{D}_v$. An assignment for a set X of variables is then a function of type $\mathcal{D}_V \rightarrow \mathcal{D}_X$.

Definition 1 (ioSTS). *An Input/Output Symbolic Transition System $\mathcal{M}$ is defined by a tuple (V, Θ, Σ, T) where:*

- $V = V_i \cup V_x$ *is the set of variables, partitioned into a set V_i of* internal *variables and a set V_x of* external *variables.*
- Θ *is the* initial condition. *It is a predicate $\Theta \subseteq \mathcal{D}_{V_i}$ defined on internal variables. It is assumed that Θ has a unique solution in $\mathcal{D}_{V_i}$.*
- $\Sigma = \Sigma_? \cup \Sigma_!$ *is the finite alphabet of* actions. *Each action a has a* signature $\mathrm{sig}(a)$, *which is a tuple of types* $\mathrm{sig}(a) = \langle t_1, \ldots, t_k \rangle$ *specifying the types of the* communication parameters *carried by the action.*
- T *is a finite set of* symbolic transitions. *A symbolic transition $t = (a, \boldsymbol{p}, G, A)$, also written $[\, a(\boldsymbol{p}) : G(\boldsymbol{v}, \boldsymbol{p})\,?\,\boldsymbol{v}'_i := A(\boldsymbol{v}, \boldsymbol{p})\,]$, is defined by*
  - *an action $a \in \Sigma$ and a tuple of* (formal) communication parameters $\boldsymbol{p} = \langle p_1, \ldots, p_k \rangle$, *which are local to a transition; without loss of generality, it is assumed that each action a always carries the same vector $\boldsymbol{p}$, which is supposed to be well-typed w.r.t. the signature* $\mathrm{sig}(a) = \langle t_1, \ldots, t_k \rangle$; *$\mathcal{D}_{\boldsymbol{p}}$ is denoted by $\mathcal{D}_{\mathrm{sig}(a)}$;*
  - *a* guard $G \subseteq \mathcal{D}_V \times \mathcal{D}_{\mathrm{sig}(a)}$, *which is a predicate on the variables (internal and external) and the communication parameters. It is assumed that guards are expressed in a theory in which satisfiability is decidable;*
  - *an assignment $A : \mathcal{D}_V \times \mathcal{D}_{\mathrm{sig}(a)} \rightarrow \mathcal{D}_{V_i}$, which defines the evolution of the internal variables. $A_v : \mathcal{D}_V \times \mathcal{D}_{\mathrm{sig}(a)} \rightarrow \mathcal{D}_v$ denotes the function in A defining the evolution of the variable $v \in V_i$.*

This model is rather standard, except for the distinction between internal and external variables. The external variables allow an ioSTS $\mathcal{M}_1$ to play the rôle of an observer (used later to formalize test purposes) by inspecting the variables of another ioSTS $\mathcal{M}_2$ when composed together with it. There is no explicit notion of control location in the model, since the control structure of an automaton can be encoded by a specific program counter variable (this will be the case in all examples).

Example 1. A simple example of ioSTS, that will serve as a running example, is described in Figure 1. The ioSTS $\mathcal{S}$ has two internal variables x and y, plus a program counter pc taking its values in $\{Rx, Ry, Cmp, End\}$. It has one input action in and three output actions end, ok and nok, and a communication parameter p. For readability, inputs are prefixed by ? and outputs by ! in examples and figures, but these marks do not belong to the alphabet.

Initially, x and y are set to 0, and pc to the location Rx. In Rx, the process either sends an output end and stops in End, or waits for an input in, carrying a value of the parameter p which is stored in x, and moves to Ry. In Ry, the value of the input parameter p of in is stored in y and the process moves to Cmp. In Cmp, either $y - x \geq 2$ and the output ok is sent with this difference, or $y - x < 2$ and nok is sent. In both cases the process loops back in Rx.

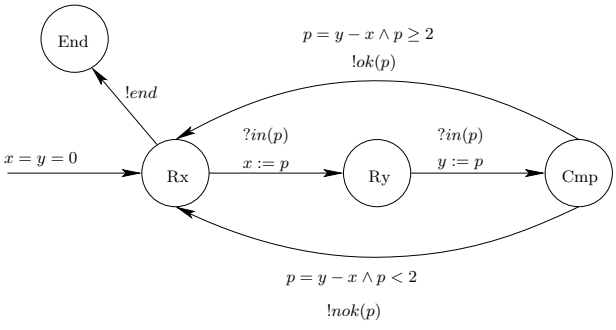

Fig. 1. ioSTS example $\mathcal{S}$

The use of external variables is motivated by the ioSTS of Figure 4 which represents an observer $\mathcal{TP}$ for $\mathcal{S}$. It has no internal variable (but could have, e.g. a counter), but has an external variable x observing the internal variable x of $\mathcal{S}$ by synchronization of $\mathcal{S}$ and $\mathcal{TP}$.

2.2 Semantics of ioSTS

The ioSTS model is a syntactic model allowing to define infinite-state transition systems. The semantics of an ioSTS is an input/output labelled transition systems (ioLTS), i.e. a labelled transition systems (LTS) with distinguished inputs and outputs. This ioLTS semantics is formally defined as follows:

Definition 2 (ioLTS semantics of an ioSTS). *The* semantics *of an ioSTS* $\mathcal{M} = (V, \Theta, \Sigma, T)$ *is an ioLTS* $[\![\mathcal{M}]\!] = (Q, Q_0, \Lambda, \rightarrow)$ *where:*

- $Q = \mathcal{D}_V$ *is the set of states;*
- $Q^0 = \{\boldsymbol{\nu} = \langle \boldsymbol{\nu}_i, \boldsymbol{\nu}_x \rangle \mid \boldsymbol{\nu}_i \in \Theta \wedge \boldsymbol{\nu}_x \in \mathcal{D}_{V_x}\}$ *is the subset of initial states;*
- $\Lambda = \{\langle a, \boldsymbol{\pi} \rangle \mid a \in \Sigma \wedge \boldsymbol{\pi} \in \mathcal{D}_{\mathrm{sig}(a)}\}$ *is the set of* valued actions *partitioned into* valued inputs $\Lambda_?$*, and* valued outputs $\Lambda_!$*;*

– *the transition relation* $\rightarrow$ *is defined by*

$$\frac{(a,\boldsymbol{p},G,A)\in T \quad \boldsymbol{\nu}=\langle \boldsymbol{\nu}_i,\boldsymbol{\nu}_x\rangle\in\mathcal{D}_V \quad \boldsymbol{\pi}\in\mathcal{D}_{\mathrm{sig}(a)} \quad \boldsymbol{\nu}'=\langle \boldsymbol{\nu}'_i,\boldsymbol{\nu}'_x\rangle\in\mathcal{D}_V \quad G(\boldsymbol{\nu},\boldsymbol{\pi}) \quad \boldsymbol{\nu}'_i=A(\boldsymbol{\nu},\boldsymbol{\pi})}{\boldsymbol{\nu}\overset{\langle a,\boldsymbol{\pi}\rangle}{\rightarrow}\boldsymbol{\nu}'} \qquad \text{(Sem)}$$

A state of the ioLTS is composed of a valuation of the internal and external variables of the ioSTS. In the initial state, the value of internal variables is uniquely defined by Θ, while the value of external variables is arbitrary. Transitions are labeled by *valued actions* composed of an action name and a valuation of communication parameters. The rule (Sem) says that a transition $(a,\boldsymbol{p},G,A)$ of an ioSTS can be fired in a state $\boldsymbol{\nu}=\langle\boldsymbol{\nu}_i,\boldsymbol{\nu}_x\rangle$, if there exists a valuation $\boldsymbol{\pi}$ of the communication parameters $\boldsymbol{p}$ such that $\langle\boldsymbol{\nu},\boldsymbol{\pi}\rangle$ satisfies the guard G; in such a case, the valued action $\langle a,\boldsymbol{\pi}\rangle$ is taken, the internal variables are assigned new values as specified by the assignment A. External variables behave similarly as *volatile* variables in the C language, by taking arbitrary values when a transition is taken. This reflects the fact that their value is defined by another ioSTS.

The semantics of an ioSTS may be an infinite-state ioLTS, because variables may have infinite domains. These ioLTS may also have infinite branching as communication parameters may also have infinite domains.

Notations, runs and traces. Given this semantics, some notions and properties of ioSTS are defined in terms of their underlying ioLTS semantics. As usual for ioLTS, $q\overset{\alpha}{\rightarrow}q'$ is used for $(q,a,q')\in\rightarrow$ and $q\overset{\alpha}{\rightarrow}$ for $\exists q', q\overset{\alpha}{\rightarrow}q'$. For a sub-alphabet $\Lambda'\subseteq\Lambda$, a state q of M is said Λ'*-complete* if $\forall\alpha\in\Lambda' : q\overset{\alpha}{\rightarrow}$. It is *complete* if it is Λ*-complete*. The ioLTS $[\![\mathcal{M}]\!]$ is Λ'*-complete* (resp. complete) if all its states are Λ'*-complete* (resp. complete). Note that these completeness conditions can be defined on ioSTS: an ioSTS $\mathcal{M}$ is Σ'-complete for $\Sigma'\subseteq\Sigma$, if for any $a\in\Sigma'$, $\bigwedge_{(a,\boldsymbol{p},G,A)\in T}\neg G$ is unsatisfiable (otherwise said $\forall a\in\Sigma', \bigvee_{(a,\boldsymbol{p},G,A)\in T} G = true$).

Using the ioLTS semantics, one can now define the behavior of an ioSTS. A *run* of an ioSTS $\mathcal{M}$ is an alternate sequence of states and valued actions $\rho = q_0\alpha_0q_1\ldots\alpha_{n-1}q_n\in Q^0.(\Lambda.Q)^*$ s.t. $\forall i, q_i\overset{\alpha_i}{\rightarrow}q_{i+1}$. For a set $F\subseteq Q$, the run ρ is *accepted in* F if $q_n\in F$. $Runs(\mathcal{M})$ denotes the set of runs of $\mathcal{M}$ and $Runs_F(\mathcal{M})$ denotes the set of accepted runs in F. When modelling the testing activity, we consider that variables and locations, thus states of the ioLTS semantics, cannot be observed by the environment. So abstractions of runs have to be considered, where states are abstracted away. A *trace* of a run $\rho\in Runs(\mathcal{M})$ is the projection $proj_\Lambda(\rho)$ of ρ on actions. $Traces(\mathcal{M})\triangleq proj_\Lambda(Runs(\mathcal{M}))$ denotes the set of traces of $\mathcal{M}$ and $Traces_F(\mathcal{M})\triangleq proj_\Lambda(Runs_F(\mathcal{M}))$ is the set of traces of runs accepted in F.

The notion of trace leads to the notion of determinism and to the determinization operation. Determinism can be defined at the syntactical level for ioSTS.

Definition 3 (Deterministic ioSTS). *An ioSTS* $\mathcal{M}=(V,\Theta,\Sigma,T)$ *is* deterministic *if for any action* $a\in\Sigma$*, and any pair of transitions* $t_1=(a,\boldsymbol{p},G_1,A_1)$

and $t_2 = (a, \boldsymbol{p}, G_2, A_2)$ *carrying the same action, the conjunction of the guards* $G_1 \wedge G_2$ *is unsatisfiable.*

Whether an ioSTS is deterministic or not can thus be decided as soon as satisfiability of guards is decidable. Note that an ioSTS $\mathcal{M} = (V, \Theta, \Sigma, T)$ with only internal variables, i.e., $V = V_i$ (this will be the case for specifications) is deterministic if and only if $[\![\mathcal{M}]\!]$ is deterministic. When the set of external variables V_x is non-empty this is not true anymore, as assignments do not constrain V_x.

For the sake of simplicity, we already restricted our attention to ioSTS with no internal actions. We focus further to deterministic ioSTS specifications. Internal actions can be handled if there is no loop of internal actions, and non-deterministic ioSTS may be handled, at least for a sub-class of ioSTS where non-determinism can be solved with bounded look-ahead. For this class, there is a determinization procedure that transforms a non-deterministic ioSTS in a deterministic one with same set of traces [19].

2.3 Visible Behaviour for Testing

During conformance testing, the tester stimulates inputs of the system under test, and observes its outputs. In testing practice, absence of output, called *quiescence*, is also observed using timers, with the assumption that timeout values are large enough such that, if a timeout occurs, the system is indeed quiescent. The tester should indeed be able to distinguish between specified and unspecified quiescence. But as trace semantics does not preserve quiescence in general, possible quiescence should be made explicit on the specification. This transforms traces into *suspension* traces [29] (i.e., traces with possible quiescence between actions). For ioLTS, the transformation, denoted Δ consists in adding a self-loop labelled with a new output δ in each quiescent state. Suspension is defined as follows for ioSTS with the expected effect on the ioLTS semantics (i.e. $[\![\Delta(\mathcal{M})]\!] = \Delta([\![\mathcal{M}]\!])$):

Definition 4 (Suspension for ioSTS). *For an ioSTS* $\mathcal{M} = (V, \Theta, \Sigma, T)$ *with alphabet* $\Sigma = \Sigma_! \cup \Sigma_?$*, the* suspension *of* $\mathcal{M}$ *is the ioSTS* $\Delta(\mathcal{M}) = (V, \Theta, \Sigma^\delta, T_\delta)$ *where*

- *the alphabet is increased by a new output:* $\Sigma^\delta = \Sigma_!^\delta \cup \Sigma_?$ *with* $\Sigma_!^\delta = \Sigma_! \cup \{\delta\}$,
- *new loop transitions labelled by* δ *are added:* $T_\delta = T \cup \{\langle \delta, G_\delta, Id_V \rangle\}$ *with*

$$G_\delta = \neg \left(\bigvee_{(a,\boldsymbol{p},G,A)\in T,\ a\in\Sigma_!} \exists \boldsymbol{\pi} \in \mathcal{D}_{\mathrm{sig}(a)} : G(\boldsymbol{\nu}, \boldsymbol{\pi}) \right)$$

G_δ evaluates to `true` when no value ν of variables and π of communication parameter can be chosen such that an output can be fired. Transition δ can thus be fired when no output can be fired, and loops in the same state[1].

[1] Note that the satisfiability of G_δ is decidable, as it is the negation of the conjunction of guards G, which satisfiability is assumed decidable.

Example 2. The suspension ioSTS of the ioSTS $\mathcal{S}$ of Figure 1 is represented in Figure 2. In this example guards of δ actions are either `true` in locations *End* and *Ry*, or `false` and discarded in other locations.

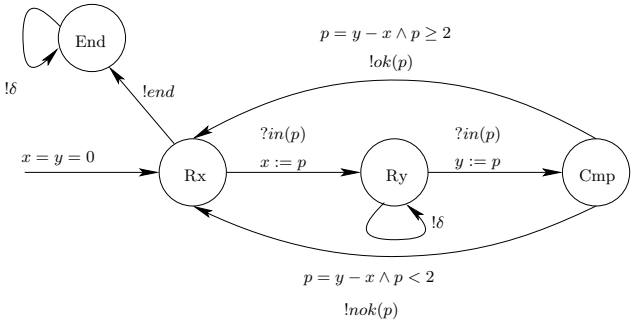

Fig. 2. Suspension ioSTS $\Delta(\mathcal{S})$

For an ioSTS $\mathcal{M}$ modelling a reactive system, the visible behaviour considered for testing is then composed of the traces of its suspension $\Delta(\mathcal{M})$. This is denoted $STraces(\mathcal{M}) \triangleq Traces(\Delta(\mathcal{M}))$. This set of *suspension traces* is considered as the reference behavior for testing conformance with respect to $\mathcal{S}$.

3 Conformance Testing Theory

The ioco testing theory of [29] can be reformulated in the context of specifications described by ioSTS. This mainly consists in precising how to model specifications, implementations and test cases, in formally defining conformance as a relation between specification and implementations, and last in modelling test executions and defining their verdicts.

Additionally, properties of test cases that relate verdicts of test executions to conformance should be required: rejection by a test case should mean non-conformance and any non-conformance should be detectable. These properties should be satisfied by the test cases which are automatically generated by our algorithms.

In terms of models for specifications, implementations and test cases, the following is assumed:

- the specification is a deterministic ioSTS $\mathcal{S} = (V^{\mathcal{S}}, \Theta^{\mathcal{S}}, \Sigma, T^{\mathcal{S}})$, with $\Sigma = \Sigma_! \cup \Sigma_?$ and $V_x^{\mathcal{S}} = \emptyset$ ($\mathcal{S}$ has only internal variables), with ioLTS semantics $[\![\mathcal{S}]\!] = S = (Q, Q^0, \Lambda, \rightarrow)$ with $\Lambda = \Lambda_! \cup \Lambda_?$.
- the implementation is modelled by a (possibly non-deterministic) ioLTS $I = (Q_I, Q_I^0, \Lambda_! \cup \Lambda_?, \rightarrow_I)$ having the same interface as $\mathcal{S}$. I is also assumed to be

$\Lambda_?$-complete[2], and let $\Delta(I)$ be its suspension ioLTS. Implementations are indeed unknown, but in order to reason about conformance, specification models need to be related to models of implementations. This is the classical test hypothesis.

- A test case for the specification ioSTS $\mathcal{S}$ is a deterministic ioSTS $\mathcal{TC} = (V^{TC}, \Theta^{TC}, \Sigma^{TC}, T^{TC})$, where $\Sigma_?^{TC} = \Sigma_!$ and $\Sigma_!^{TC} = \Sigma_?$ (actions are mirrored w.r.t. $\mathcal{S}$), equipped with a variable $\mathsf{Verdict} \in V^{TC}$ of the enumerated type $\{\texttt{none}, \texttt{fail}, \texttt{pass}, \texttt{inconc}\}$. Intuitively, `fail` means rejection, `pass` means that some targeted behaviour has been reached (this will be clarified later) and `inconc` means that targeted behaviours cannot been reached anymore. $\mathcal{TC}$ is assumed to be $\Sigma_?^{TC}$-complete in all states where $\mathsf{Verdict} = \texttt{none}$. This means that $\mathcal{TC}$ is ready to react to any output of the implementation, except when a verdict is reached and the execution stops. Let $TC = [\![\mathcal{TC}]\!] = (Q^{TC}, q_0^{TC}, \Lambda^{TC}, \rightarrow_{TC})$ denote the ioLTS semantics of $\mathcal{TC}$. One denotes by $\mathsf{Fail} = (\mathsf{Verdict} = \texttt{fail})$, $\mathsf{Pass} = (\mathsf{Verdict} = \texttt{pass})$, and $\mathsf{Inconc} = (\mathsf{Verdict} = \texttt{inconc})$ the subsets of Q^{TC} where verdicts are emitted.

Conformance relation. In this setting, a conformance relation defines the set of correct ioLTS implementations I of an ioSTS specification $\mathcal{S}$. In this paper, the usual ioco relation of Tretmans [29] is considered. This relation defines conformance as a partial inclusion of suspension traces. A definition which is equivalent to the original one can be given as follows.

Definition 5 (Conformance). *Let I be an implementation and $\mathcal{S}$ a specification of I. The* ioco *conformance relation is defined as:*

$$I \text{ ioco } \mathcal{S} \triangleq STraces(I) \cap NC\_STraces(\mathcal{S}) = \emptyset$$

where $NC\_STraces(\mathcal{S}) = STraces(\mathcal{S}) \cdot (\Lambda_! \cup \{\delta\}) \setminus STraces(\mathcal{S})$.

The set of traces $NC\_STraces(\mathcal{S})$ thus exactly characterizes the set of non-conformant behaviours: I is non-conformant as soon as it may exhibit a suspension trace of $\mathcal{S}$ extended with an unspecified output or unexpected quiescence. Interestingly, this formulation of ioco explicits the fact that conformance to a given specification is indeed a safety property of I in the usual meaning: conformance w.r.t. $\mathcal{S}$ is violated if I exhibits a finite trace in $NC\_STraces(\mathcal{S})$.

It is possible to characterize $NC\_STraces(\mathcal{S})$ by an ioSTS observer accepting exactly this set of traces. This ioSTS called *canonical tester* is built from $\Delta(\mathcal{S})$ as follows:

Definition 6 (Canonical Tester). *Let $\mathcal{S} = (V^{\mathcal{S}}, \Theta^{\mathcal{S}}, \Sigma, T^{\mathcal{S}})$ be a deterministic ioSTS for the specification and $\Delta(S) = (V^{\mathcal{S}}, \Theta^{\mathcal{S}}, \Sigma^{\delta}, T_{\delta}^{\mathcal{S}})$ its suspension. The* canonical tester *for $\mathcal{S}$ is the (deterministic) ioSTS $Can(\mathcal{S}) = (V^{Can}, \Theta^{Can}, \Sigma^{Can}, T^{Can})$ such that*

[2] This ensures that the composition of I with a test case TC never blocks because of non-implemented inputs.

- $V^{Can} = V^{\mathcal{S}} \cup \{\mathsf{Verdict}\}$ *where* Verdict *is of the enumerated type* $\{\mathtt{none}, \mathtt{fail}\}$
- $\Theta^{Can} = \Theta^{\mathcal{S}} \wedge \mathsf{Verdict} = \mathtt{none}$*;*
- $\Sigma_?^{Can} = \Sigma_!^{\delta}$ *and* $\Sigma_!^{Can} = \Sigma_?$ *(the input-output alphabet is mirrored w.r.t.* $\Delta(\mathcal{S})$*)*
- T^{Can} *is defined by the rules:*

$$\frac{t \in T^{\mathcal{S}}}{t \in T^{Can}} \qquad \text{(Keep } T^{\mathcal{S}}\text{)}$$

$$\frac{a \in \Sigma_!^{\delta} = \Sigma_?^{Can} \quad G_a = \bigwedge_{(a,\boldsymbol{p},G,A)\in T^{\mathcal{S}}} \neg G}{\left[\, a(\boldsymbol{p}) \;:\; G_a(\boldsymbol{v},\boldsymbol{p}) \,?\, \mathsf{Verdict}' := \mathtt{fail} \,\right] \in T^{Can}} \qquad \text{(Input-completion)}$$

It is easy to see that $Can(\mathcal{S})$ is a test case by itself: it is deterministic and $\Sigma_?^{Can}$-complete in all states where Verdict = none. Moreover it exactly characterizes non-conformant behaviours as $Traces_{\mathsf{Fail}}(Can(\mathcal{S})) = NC\_STraces(\mathcal{S})$. Consequently conformance can be written:

$$I \;\mathsf{ioco}\; \mathcal{S} \iff STraces(I) \cap Traces_{\mathsf{Fail}}(Can(\mathcal{S})) = \emptyset$$

If I was known, verifying conformance could be reduced to a reachability problem: check whether Fail is reachable in the synchronous product of ioLTS $I \times [\![Can(\mathcal{S})]\!]$. This may be difficult when $[\![Can(\mathcal{S})]\!]$ is infinite.

However, as I is unknown, one can only make experiments with selected test cases, providing inputs and checking that outputs and quiescences of I are specified in $\mathcal{S}$. This entails that, except in simple cases, conformance cannot be proved by testing, only non-conformance witnesses can be exhibited.

Example 3. The Figure 3 represents an abstract view of the canonical tester of the example specification $\mathcal{S}$ of Figure 1. For example, in location *Cmp*, there should be a transition labelled by the input ?*ok* with guard $p \neq y - x \vee p < 2$, a transition labelled by the input ?*nok* with guard $p \neq y - x \vee p \geq 2$ and a transition labelled by the input ?*end* with guard true, all these transitions having the assignment Verdict := fail. Rather than explicitly describing guards of added transitions, all these transitions are represented by a single transition with label ?*otherwise* and target location Fail from the meta-location composed of all locations.

Test execution. The execution of a test case on an implementation is now considered. This execution is naturally modelled as a composition of processes. As quiescence of I is observed during testing, the composition is with $\Delta(I)$ which explicits this quiescence. Formally, the execution of a test case $\mathcal{TC}$ on an implementation I is modelled by the *parallel composition* of $TC = [\![\mathcal{TC}]\!]$ with $\Delta(I)$ with synchronization on common actions.

Definition 7 (Test execution). *Let* $\Delta(I) = (Q^I, Q_0^I, \Lambda_! \cup \{\delta\} \cup \Lambda_?, \rightarrow_{\Delta(I)})$ *and* $TC = (Q^{TC}, q_0^{TC}, \Lambda_? \cup \Lambda_! \cup \{\delta\}, \rightarrow_{TC})$*. The test execution of* TC *on the implementation* I *is modelled by the parallel composition of* $\Delta(I)$ *and* TC *, which*

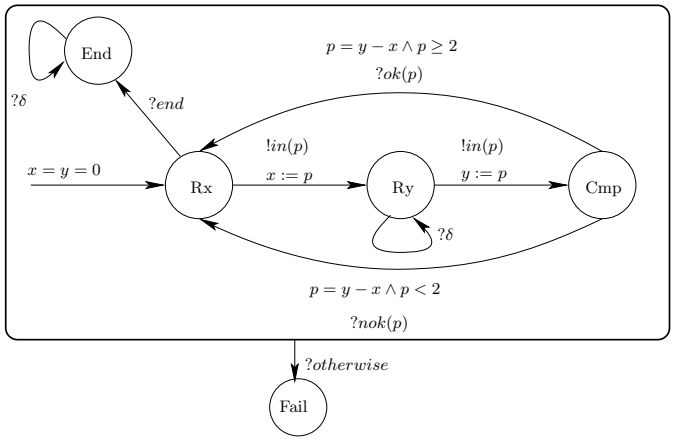

Fig. 3. Canonical tester $Can(\mathcal{S})$

is the ioLTS $\Delta(I) \| TC = (Q^I \times Q^{TC}, Q_0^I \times \{q_0^{TC}\}, \Lambda_! \cup \{\delta\} \cup \Lambda_?, \rightarrow_{\Delta(I)\| TC})$ *where* $\rightarrow_{\Delta(I)\| TC}$*, is defined by the rule:*

$$\frac{\alpha \in \Lambda_! \cup \{\delta\} \cup \Lambda_? \quad q_1 \xrightarrow{\alpha}_{\Delta(I)} q_2 \quad q_1' \xrightarrow{\alpha}_{TC} q_2'}{(q_1, q_1') \xrightarrow{\alpha}_{\Delta(I)\| TC} (q_2, q_2')}$$

With this definition, it is clear that $Traces(\Delta(I) \| TC) = STraces(I) \cap Traces(TC) = STraces(I) \cap Traces(\mathcal{TC})$. For $P \in \{\mathsf{Fail}, \mathsf{Pass}, \mathsf{Inconc}\}$, one also has $Traces_{Q^I \times P}(\Delta(I) \| TC) = STraces(I) \cap Traces_P(TC)$.

A test case rejects an implementation when the Verdict variable reaches the value `fail`. The possible rejection of I by TC is then defined by the fact that $\Delta(I) \| TC$ may lead to Fail in TC: *TC mayfail I* $\triangleq Traces_{Q^I \times \mathsf{Fail}}(\Delta(I) \| TC) \neq \emptyset$ which is equivalent to $STraces(I) \cap Traces_{\mathsf{Fail}}(TC) \neq \emptyset$. Similar definitions can be given for *maypass* and *mayinconc*.

Test case properties. Test generation/selection algorithms should produce test cases with properties relating rejection with non-conformance. Formally,

Definition 8 (Soundness, exhaustiveness). *Let TS be a set of test cases. TS is said* complete *if it is both* sound *and* exhaustive *where:*

- *TS is* sound $\triangleq \forall I : (I \text{ ioco } S \implies \forall TC \in TS : \neg(TC \text{ mayfail } I))$*, i.e., only non-conformant implementations can be rejected by a test case in TS.*
- *TS is* exhaustive $\triangleq \forall I : (\neg(I \text{ ioco } S) \implies \exists TC \in TS : TC \text{ mayfail } I)$*, i.e., any non-conformant implementation can be rejected by a test case in TS.*

Using the facts that I ioco S is equivalent to $STraces(I) \cap Traces_{\mathsf{Fail}}(Can(\mathcal{S})) = \emptyset$ and that *TC mayfail I* is equivalent to $STraces(I) \cap Traces_{\mathsf{Fail}}(TC) \neq \emptyset$, one can prove the following properties:

Proposition 1. *Let TS be a set of test cases for the specification* $\mathcal{S}$*,*
TS is sound iff $\bigcup_{TC \in TS} Traces_{\mathsf{Fail}}(TC) \subseteq Traces_{\mathsf{Fail}}(Can(\mathcal{S}))$*,*
TS is exhaustive iff $\bigcup_{TC \in TS} Traces_{\mathsf{Fail}}(TC) \supseteq Traces_{\mathsf{Fail}}(Can(\mathcal{S}))$*.*

The proposition says that sound test cases are sub-observers of $Can(\mathcal{S})$, and that an exhaustive test suite must reject all implementations rejected by $Can(S)$, and thus should cover all these non-conformance detections. It immediately follows that the canonical tester $Can(\mathcal{S})$ alone forms a complete test suite. In some sense $Can(\mathcal{S})$ is the most general testing process for conformance w.r.t. $\mathcal{S}$.

Taking a close look at test generation algorithms for ioco which define complete test suites, one notice that all these algorithms produce an infinite number of unfoldings of $Can(S)$ covering all Fail traces.

Despite the nice properties of $Can(S)$, in practice it cannot be used directly as a test case. In fact, one wants to select individual test cases focused on some particular behaviour. In particular one often avoids the choice between two outputs in a test case, as outputs are controllable by the tester. Selection of a sound test suite will then be based on the selection of sub-behaviours of $Can(S)$. A consequence of this selection is that exhaustiveness is often lost if only a finite number of test cases is selected. However, the selection algorithm should remain *limit exhaustive*: for any non-conformant implementation, the algorithm should be able to select a test case that could reject this implementation. In other words, the infinite set of test cases that could be selected should be exhaustive. This is important as this guarantees that any non-conformance is detectable. The contrary would mean that the selection algorithm is too weak, or that the conformance relation is too strong compared to the capability of test cases to distinguish between conformant and non-conformant behaviors.

4 Test Selection for ioSTS

In this section, the selection of ioSTS test cases from an ioSTS specification $\mathcal{S}$ is explained. As explained previously, test selection consists in extracting a sub-observer of the non-conformance observer $Can(\mathcal{S})$. The first step consists in constructing the ioSTS $Can(\mathcal{S})$, using Definition 6.

4.1 Test Purposes and Test Selection Problem

Several means have been investigated for test selection including random or non-deterministic generation, test purposes, coverage criteria, etc. This paper focuses on the selection of test cases by test purposes. Intuitively, a test purpose describes some abstract behaviours one wants to test. A test purpose is here formally defined as an ioSTS equipped with a set of accepting locations playing the role of a non-intrusive observer.

Definition 9 (Test purpose). *A* Test Purpose *for a specification ioSTS* $\mathcal{S} = (V, \Theta, \Sigma = \Sigma_! \cup \Sigma_?, T)$*, is a deterministic ioSTS* $\mathcal{TP} = (V^{TP}, \Theta^{TP}, \Sigma^{\delta}, T^{TP})$ *such that*

- $V_x^{TP} = V_i^S$*: test purposes are allowed to observe the internal state of* $\mathcal{S}$*;*
- $V_i^{TP} \cap V_i^S = \emptyset$ *and* V_i^{TP} *contains a program counter variable* pc^{TP} *with* $\texttt{accept} \in \mathcal{D}_{pc^{TP}}$*. Its set of accepting states is denoted by* $\mathsf{Accept} = (pc^{TP} = \texttt{accept})$*.*
- $\mathcal{TP}$ *should be* complete *except when* $pc^{TP} = \texttt{accept}$*, which means that for any action* $a \in \Sigma^\delta$*,* $pc^{TP} \neq \texttt{accept} \Rightarrow \bigvee_{(a,\boldsymbol{p},G,A)\in T^{TP}} G = true$*. This ensures that* $\mathcal{TP}$ *does not restrict the runs of* $\mathcal{S}$ *before they are accepted (if ever).*

Example 4. An example of test purpose is described in Figure 4. It specifies that one wants to select behaviors of $\mathcal{S}$ ending with $ok(2)$ when the value of x is greater than 3, without any output nok and no output $ok(p)$ with $p \neq 2$ or when $x < 3$. The label "*" is used for completion, and means "any action with guard being the negation of the conjunction of guards on specified transitions carrying this action". This test purpose describes behaviors in an abstract way since one can focus on some actions of interest without explicitly specifying intermediate ones.

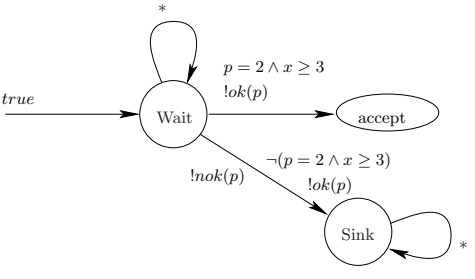

Fig. 4. ioSTS test purpose $\mathcal{TP}$

The role of a test purpose is to select runs of $Can(\mathcal{S})$ accepted by $\mathcal{TP}$. Remember that runs are languages where both actions and states have meanings. The usual way to define such an intersection of languages in the case of ioLTS, is to perform a *synchronous product*. This can be extended to runs by synchronizing states. An operation with similar effect on the ioLTS semantics, can be defined on ioSTS, thus defined at a syntactical level, where transitions with same actions synchronize on the conjunction of their guards, and synchronization of states is preformed by the observation of external variables. A general definition could be given, but it is specialized here to the product of the canonical tester of a specification and a test purpose, for its use in test selection. Formally,

Definition 10 (Synchronous product of ioSTS). *Let* $Can(\mathcal{S}) = (V^{Can}, \Theta^{Can}, \Sigma^\delta, T^{Can})$ *be the canonical tester of* $\mathcal{S}$ *and* $\mathcal{TP} = (V^{TP}, \Theta^{TP}, \Sigma^\delta, T^{TP})$ *a test purpose with* $V_x^{TP} = V_i^{Can}$*. The* synchronous product *of* $Can(\mathcal{S})$ *and* $\mathcal{TP}$ *is the ioSTS* $\mathcal{P} = Can(\mathcal{S}) \times \mathcal{TP} = (V^P, \Theta^P, \Sigma^{Can}, T^P)$ *where*

– $V^P = V_i^P \cup V_x^P$, *with* $V_i^P = V_i^{Can} \cup V_i^{TP}$ *and* $V_x^P = \emptyset$;
– $\Theta^P(\langle \boldsymbol{v}^{Can}, \boldsymbol{v}^{TP} \rangle) = \Theta^{Can}(\boldsymbol{v}^{Can}) \wedge \Theta^{TP}(\boldsymbol{v}^{TP})$;
– T^P *is defined by the following inference rule:*

$$\frac{[\, a(\boldsymbol{p}) \,:\, G^c(\boldsymbol{v}^c, \boldsymbol{p})\,?\,(\boldsymbol{v}_i^c)' := A^c(\boldsymbol{v}^c, \boldsymbol{p})\,] \in T^{Can} \quad [\, a(\boldsymbol{p}) \,:\, G^t(\boldsymbol{v}^t, \boldsymbol{p})\,?\,(\boldsymbol{v}_i^t)' := A^t(\boldsymbol{v}^t, \boldsymbol{p})\,] \in T^{TP}}{\begin{array}{c}[a(\boldsymbol{p}) \,:\, G^c(\boldsymbol{v}^c, \boldsymbol{p}) \wedge G^t(\boldsymbol{v}^t, \boldsymbol{p})\,? \\ (\boldsymbol{v}_i^c)' := A^c(\boldsymbol{v}^c, \boldsymbol{p}), (\boldsymbol{v}_i^t)' := A^t(\boldsymbol{v}^t, \boldsymbol{p})] \in T^P\end{array}}$$

Let $\mathcal{P}'$ *be the ioSTS obtained by adding the assignment* Verdict := pass *to all transitions with assignment* pc' := accept.

Example 5. The synchronous product of $Can(\mathcal{S})$ and $\mathcal{TP}$ for our running example is (partly) described in Figure 5. Note for example the synchronization on the input *ok* of guards $p = y - x \wedge p \geq 2$ of $\mathcal{S}$ with $p = 2 \wedge x \geq 3$ from $\mathcal{TP}$.

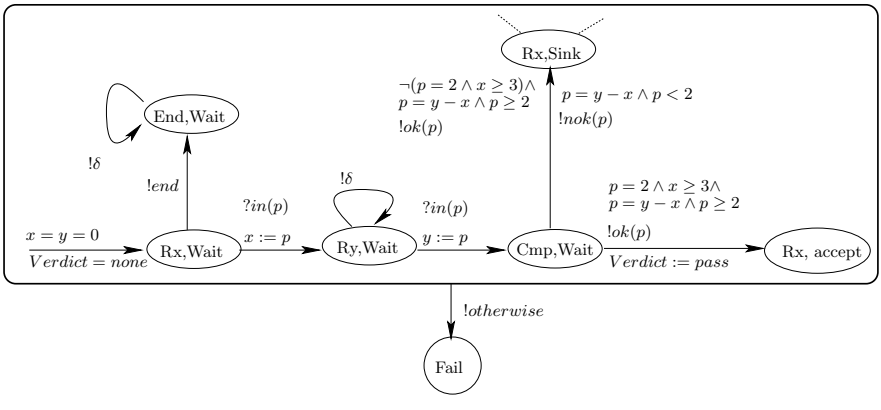

Fig. 5. Synchronous product $Can(\mathcal{S}) \times \mathcal{TP}$

As $\mathcal{TP}$ is non-intrusive (it observes but does not modify $\mathcal{S}$ variables), one gets $Traces(\mathcal{P}') \subseteq Traces(Can(\mathcal{S}))$ and $Traces_{\mathsf{Fail}}(\mathcal{P}') = Traces(\mathcal{P}') \cap Traces_{\mathsf{Fail}}(Can(\mathcal{S}))$. This means that $\mathcal{P}'$ detects every non-conformance along its traces. It is thus a sound test case.

One also has $Traces_{\mathsf{Pass}}(\mathcal{P}') = Traces_{\mathsf{Accept}}(\mathcal{P}) \subseteq STraces(\mathcal{S}) \cap Traces_{\mathsf{Accept}}(\mathcal{TP})$. This inclusion on traces comes from an equality on runs, lost by projection: even if a run of $Can(\mathcal{S})$ has the same trace as an accepted run, it may be not accepted by $\mathcal{TP}$ because of a condition on its variables observed by $\mathcal{TP}$. Depending on the considered distinguished states Fail or Pass, the ioSTS observer $\mathcal{P}'$ is both an observer of non-conformant traces and an observer of traces of accepted runs.

It has been shown that $\mathcal{P}'$ is a sound test case. However, as it is an unfolding of $Can(\mathcal{S})$, no selection has been performed yet. Selection is now needed by

focusing on traces accepted in Pass. Ideally, one would like to select exactly $Traces_{\mathsf{Pass}}(\mathcal{P}')$, plus unspecified outputs prolonging prefixes of these traces into Fail (i.e., traces in $NC\_STraces(S)$), denoting detection of non-conformance.

However, the implementations which are considered, even if they are deterministic, are *non-controllable*: their output behaviour is not completely determined by inputs. Thus, after a trace, the tester should consider all possible outputs: those from which Pass is reachable or Fail is reached, but also those after which Pass is not reachable anymore. In this last case, this divergence should be detected as soon as possible, and the Inconc verdict should be set.

This reduces to the problem of computing the set $coreach(\mathsf{Pass})$ of states from which Pass is reachable. This set can be described by a least fix-point: $coreach(\mathsf{Pass}) = \mathrm{lfp}(\lambda X.\mathsf{Pass} \cup pre(X))$ where $pre(X) = \{q \mid \exists q' \in X, \exists \alpha \in \Lambda : q \xrightarrow{\alpha} q'\}$ is the set of states from which X can be reached in one transition. The computation of $coreach(\mathsf{Pass})$ is easy for finite-state systems and can be solved with graph algorithms, as is done in the context of test selection for (finite-state) ioLTS by the TGV tool [16]. However, $coreach(\mathsf{Pass})$ is not computable for *ioSTS* models which have an infinite-state ioLTS semantics. Coping with this computability problem is the subject of the next subsection.

4.2 Approximate Analysis for Test Selection

Faced to this non-computability problem, the proposed solution consists in relying on an over-approximate co-reachability analysis. Using this approximate analysis, the ioSTS $\mathcal{P}'$ is transformed into a test case ioSTS $\mathcal{TC}$ by constraining outputs and detecting inconclusive inputs using syntactical transformations of guards of transitions.

Let $\mathcal{P}' = (V^{P'}, \Theta^{P'}, \Sigma^{Can}, T^{P'})$ as defined by Def. 10. Assume that an over-approximation $coreach^{\alpha} \supseteq coreach(\mathsf{Pass})$ of the exact set of states co-reachable from Pass has been computed. It is assumed that $coreach^{\alpha}$ is represented by a logical formula.

Moreover, given a set of states $X \in \mathcal{D}_{\boldsymbol{v}}$ represented by a formula $X(\boldsymbol{v})$, let $pre(A)(X)(\boldsymbol{v}, \boldsymbol{p})$ denote the precondition of X by an assignment $A : \mathcal{D}_{\boldsymbol{v}} \times \mathcal{D}_{\boldsymbol{p}} \to \mathcal{D}_{\boldsymbol{v}}$:

$$pre(A)(X)(\boldsymbol{v}, \boldsymbol{p}) \;=\; \exists \boldsymbol{v'} : X(\boldsymbol{v'}) \wedge \boldsymbol{v'} = A(\boldsymbol{v}, \boldsymbol{p}) \;=\; X(A(\boldsymbol{v}, \boldsymbol{p}))$$

In other words, $pre(A)(X)(\boldsymbol{v}, \boldsymbol{p})$ represents the set of values of variables $\boldsymbol{v}$ and parameters $\boldsymbol{p}$ from which X is reached after the assignment A. Note that the operator $pre(A)$ is monotone. Let $pre^{\alpha}(A)(X) \supseteq pre(A)(X)$ denote a monotone over-approximation of $pre(A)(X)$.

In this context, $pre^{\alpha}(A)(coreach^{\alpha})$ is an over-approximation of the set of values for variables and parameters which allow to stay in $coreach(\mathsf{Pass})$ when taking a transition $(a, \boldsymbol{p}, G, A)$, or in other words it is a *necessary condition* to stay in $coreach(\mathsf{Pass})$. Its negation is thus a *sufficient condition* to leave $coreach(\mathsf{Pass})$.

Using these approximate analyses, and the remarks above, a test case can be constructed from $\mathcal{P}'$ as follows. The test case for $\mathcal{S}$ and $\mathcal{TP}$ is the ioSTS

$\mathcal{TC} = (V^{P'}, \Theta^{P'}, \Sigma^{Can}, T^{\mathcal{TC}})$ where $T^{\mathcal{TC}}$ is defined by

$$\frac{(a, \boldsymbol{p}, G, A) \in T^{P'} \quad a \in \Sigma^{Can}_{!} \quad G' = pre^{\alpha}(A)(coreach^{\alpha})}{(a, \boldsymbol{p}, G \wedge G', A) \in T^{\mathcal{TC}}} \quad \text{(Select)}$$

$$\frac{(a, \boldsymbol{p}, G, A) \in T^{P'} \quad a \in \Sigma^{Can}_{?} \quad A_{\mathsf{Verdict}} = \mathsf{Verdict}' := \texttt{fail}}{(a, \boldsymbol{p}, G, A) \in T^{\mathcal{TC}}} \quad \text{(Fail)}$$

$$\frac{(a, \boldsymbol{p}, G, A) \in T^{P'} \quad a \in \Sigma^{Can}_{?} \quad A_{\mathsf{Verdict}} \neq \mathsf{Verdict}' := \texttt{fail} \quad G' = pre^{\alpha}(A)(coreach^{\alpha})}{(a, \boldsymbol{p}, G \wedge G', A), (a, \boldsymbol{p}, G \wedge \neg G', A') \in T^{\mathcal{TC}}} \quad \text{(Split)}$$

where A' is defined by $\begin{cases} A'_{\mathsf{Verdict}} = \mathsf{Verdict}' := \texttt{inconc}, \\ A'_v = A_v \text{ for } v \neq \mathsf{Verdict}, \end{cases}$

The rule (Select) constrains the guards of all *output* transitions such that their post-conditions lead to $coreach^{\alpha}$, which is an over-approximation of the set of states leading to `pass`. The intuition is that the tester controls its output transitions, thus may restrict them in $G' = pre^{\alpha}(A)(coreach^{\alpha})$ so as to have a chance to stay in the over-approximate co-reachable state-space $coreach^{\alpha}$ after the transition is fired. The rule (Fail) keeps *input* transitions leading to the `fail` verdict. The rule (Split) is illustrated by Figure 6. The rule splits the *input* transitions not leading to Fail using a conjunction with guards G' and $\neg G'$: for values of variables and parameters that certainly do not lead to $coreach^{\alpha}$ (i.e., when $\neg G'$ is true), the `inconc` verdict is emitted, while for values of variables and parameters that may lead to $coreach^{\alpha}$ (i.e., when G is true) nothing is changed. The intuition is that input transitions cannot be controlled, but the bad situations from which the verdict `pass` is not reachable may still be detected. In such a case, the verdict `inconc` is emitted. Note that the ioLTS semantics of $\mathcal{TC}$ is different from the semantics of $\mathcal{P}'$, in particular some output transitions have been removed by rule (Select).

The test case can be further simplified (but without modifying its semantics) with an over-approximation $reach^{\alpha}(\Theta^{P'})$ of its reachable states $reach(\Theta^{P'})$ where $reach(\Theta^{P'}) = \mathrm{lfp}(\lambda X.\Theta^{P'} \cup post(X))$ with $post(X) = \{q' \mid \exists q \in X, \exists \alpha \in \Lambda : q \xrightarrow{\alpha} q'\}$ being the set of states reachable from X in one transition. The simplification consists in removing transitions of which the guards are unsatisfiable in the over-approximation $reach^{\alpha}(\Theta^{P'})$ of the set of reachable states i.e., transitions $(a, \boldsymbol{p}, G, A)$ where $G \wedge reach^{\alpha}(\Theta^{P'})$ simplifies to `false`.

Example 6. The computation of $coreach^{\alpha}$ for our running example is described in Figure 7 where the formula is split in boxes attached to each location. The resulting test case, obtained after this co-reachability analysis is represented in Figure 8 (in this figure formulas in boxes will be considered later). In this simple case, an over-approximate analysis based on polyhedra gives an exact result. The effect of the analysis and application of rules is to constrain the guard of the first output *in* with $p \geq 3$ (thus to remove the controllable output *in* with guard $p < 3$ as this certainly leads outside $coreach(\mathsf{Pass})$), and to constrain the

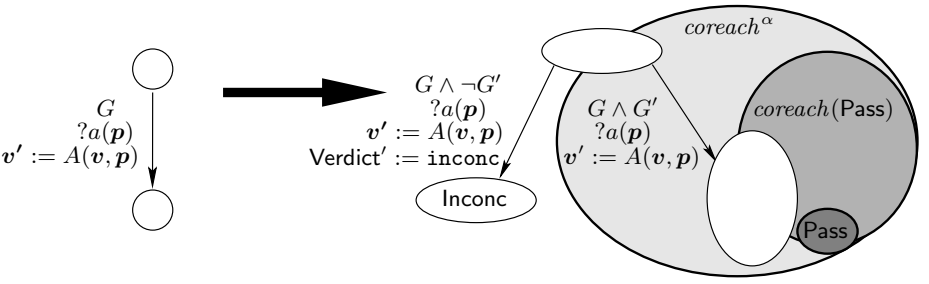

Fig. 6. Illustration of the rule (Split)

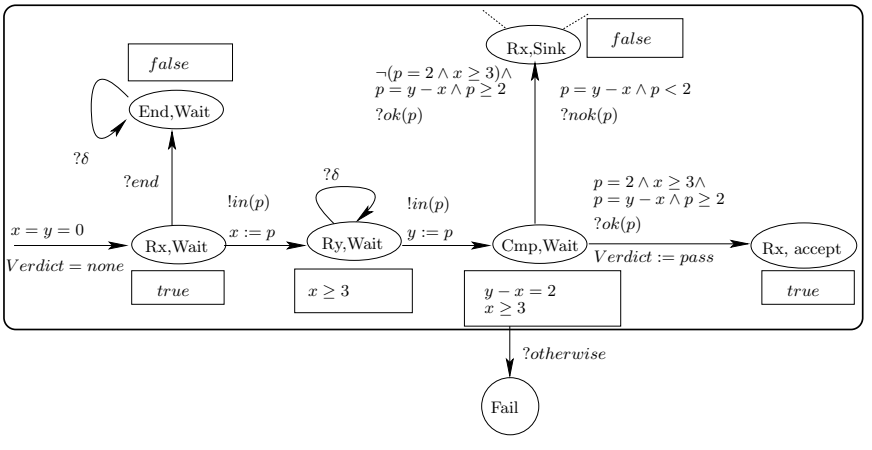

Fig. 7. Computation of $coreach^{\alpha}$

second output *in* with $p = x + 2$ (thus remove the controllable output *in* with $p \neq x + 2$ as this certainly leads outside $coreach(\mathsf{Pass})$).

A reachability analysis on the resulting test case (in boxes attached to locations in Figure 8) allows to further simplify the test case into the one represented in Figure 9). The two transitions from $Cmp, Wait$ to Inconc can be removed: in fact reachability in $Cmp, Wait$ gives $y - x = 2 \wedge x \geq 3$, thus the guard of the transition labelled by *nok* ($p = y - x \wedge p < 2$) simplifies to `false` in this context, as well as the guard of *ok*. This illustrates the fact that constraining the guards of outputs using an over-approximate co-reachability analysis can suppress some paths not leading to Pass.

In such a case where the analysis is exact, the resulting test case is optimal (but not perfect) with respect to its ability to force the reachability to Pass. Nevertheless some uncontrollable actions leading to Inconc may persist: this is the case in our example for the *end* input which cannot be avoided. A less accurate approximation would give a less selected test case with more possibilities to

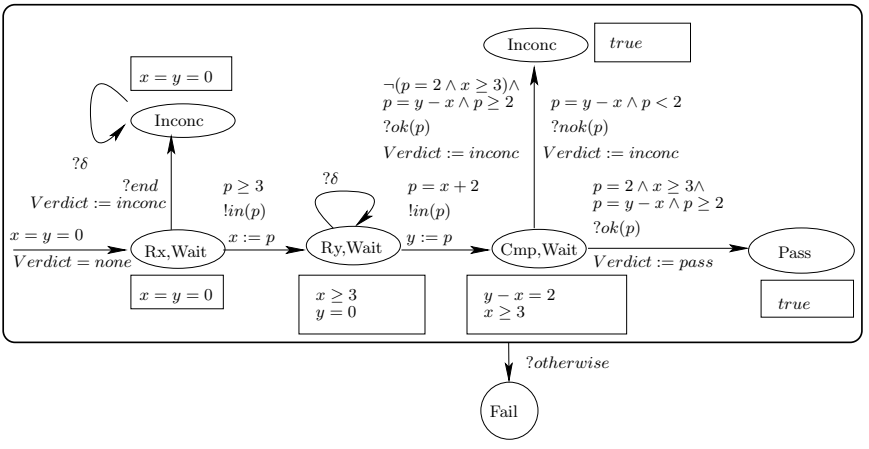

Fig. 8. Resulting test case $\mathcal{TC}$ and computation of $reach(\Theta^{P'})$

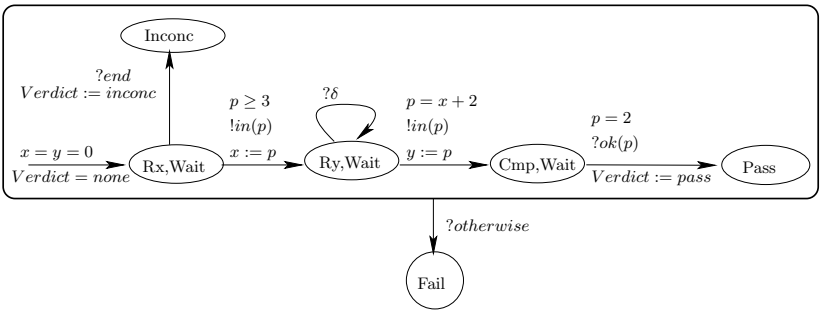

Fig. 9. Resulting test case $\mathcal{TC}$ after simplification

reach Inconc with *ok* or *nok*. For example, an analysis which ignores the values of variables would not enforce guards at all, giving as test case the ioSTS P' of Figure 7 with transitions leading to $End, Wait$ and $Rx, Sink$ producing Inconc.

4.3 Test Case Properties

It has already been proved that $Can(S)$ is sound. It is also easy to see that this property is preserved by the synchronous product $\mathcal{P}'$ and selection of $\mathcal{TC}$, as these transformations cannot add any case of rejection. Thus all test cases are sound. Moreover, one can prove the stronger property $Traces_{\mathsf{Fail}}(\mathcal{TC}) = Traces(\mathcal{TC}) \cap Traces_{\mathsf{Fail}}(Can(\mathcal{S}))$, which is preserved from the same property applied to $\mathcal{P}'$. This property says that only non-conformant implementations can be rejected, and that this rejection happens as soon as possible.

Limit exhaustiveness comes from the following construction: by definition of ioco, for any non-conformant implementation I, there exists a trace $\sigma.a$ in $STraces(I) \cap NC\_STraces(\mathcal{S})$. The prefix σ is thus a trace of $\Delta(\mathcal{S})$, while $\sigma.a \in Traces_{\mathsf{Fail}}(Can(\mathcal{S}))$. As $\Delta(\mathcal{S})$ has no quiescence ($\Delta(\Delta(\mathcal{S})) = \Delta(\mathcal{S})$), there exists an output b such that $\sigma.b$ is in $STraces(\mathcal{S})$ and thus in $Traces(Can(\mathcal{S}))$ but not in $Traces_{\mathsf{Fail}}(Can(\mathcal{S}))$. It then suffices to construct a test purpose $\mathcal{TP}$ such that the trace $\sigma.b$ leads to Accept. Now, let $\mathcal{TC}$ be the test case obtained from $\mathcal{S}$ and $\mathcal{TP}$. One then gets $\sigma.a \in STraces(I) \cap Traces_{\mathsf{Fail}}(\mathcal{TC})$ which by definition means that $\mathcal{TC}$ may reject I.

Soundness and exhaustiveness restrict properties to the relation of Fail verdicts to conformance. When using test purposes however, one is also interested in properties of test case verdicts Pass and Inconc w.r.t. the test purposes. This is where over-approximation has an influence. It is perhaps surprising to see that Pass verdicts are always exact in the following sense: $Traces_{\mathsf{Pass}}(\mathcal{TC}) = Traces(\mathcal{TC}) \cap Traces_{\mathsf{Accept}}(\mathcal{P})$. What is lost by the over-approximation of $coreach(\mathsf{Accept})$, compared with an (hypothetical) exact computation, is the ability to provide the most adequate inputs to the implementation, and the ability to detect infeasible traces to Accept as soon as this happens, thus to give Inconc verdicts as soon as possible. A detailed study of the influence of the precision of the analysis on the accuracy of test cases is presented in [18]. It is not surprising that the more precise the approximation is, the more accurate test cases are.

4.4 Test Execution

Test cases produced so far are ioSTS. In particular the values of communication parameters of test cases are not instantiated. During test execution, values of communication parameters have to be chosen for outputs of the test cases, among values satisfying the guard (e.g. $p = 5$ for $p \geq 3$ in the example). This is performed by a constraint solver. Conversely, when receiving an input from the implementation, or when observing quiescence, as the test case is input complete and deterministic, one has to check which transition can be fired, by checking the guard with the value of the received communication parameter (e.g. go to Pass if $p = 2$, and Fail otherwise).

4.5 The STG Tool

The principles of test selection described in this paper are implemented in a new version of the STG tool [8] (see `http://www.irisa.fr/vertecs/software.html#STG`). STG implements the main operations needed for selection: the synchronous product and test selection. This selection is based on approximate co-reachability and reachability analyses. These analyses are provided by an interface with the NBac tool [17] using abstract interpretation [9].

Notice that these analyses can be improved using the dynamic partitioning facility of NBac, allowing to separate locations with respect to the analysis according to some criteria. This has proved very useful in some case studies.

5 Related Work

Recently, attempts have been made to generate test cases from models of reactive systems with data [26,10,11]. The challenge here is to generate test cases without enumerating their state space, but rather by working directly on the higher-level specification models, and thus avoiding the state-space explosion problem.

These models, whether they are called extended finite-state machines or symbolic transition systems are essentially automata that manipulate variables (integers, booleans, aggregate types, ...) and correspond to programs without recursive procedure calls and without memory allocation. Testing theories stay unchanged, being based on the semantics of the models in terms of (infinite) transition systems. But the algorithms must be adapted to cope with data in a symbolic way.

Some pioneering approaches were based on extended finite-state machines (EFSM) (see e.g. [20]), but the data and control parts were mostly treated separately. An exception is the work of [26] in which the authors explore the problem of generating confirming configuration sequences that distinguish a configuration (global state) with a set of configurations, on an EFSM model. They use product of machines, projections on variables and model-checking techniques to derive such sequences.

[10] is an attempt to lift the ioco testing theory of LTS to finitely branching Symbolic Transition Systems (STS). In STS, data are specified by algebraic data types. This paper proposes an on-the-fly test generation algorithm à la TorX, based on this specification model. This formalization however does not avoid the (partial) enumeration of the LTS semantics of STS.

Several approaches are based on symbolic execution [5,14] and constraint resolution. In [12], the principle of the DART tool is to combine symbolic execution of a program with random testing. Starting from a random test case, a symbolic execution is computed on the program for this execution, giving rise to a new test case by negating the last condition of the symbolic execution path. By repeating this principle, the main execution paths are covered. The PET tool [13] also uses constraint solving to produce test cases as solutions to path conditions produced from a flow-chart specification and an extended automaton specifying a property. In [11] the authors use symbolic execution on ioSTS specification models to build a symbolic execution tree representing all behaviors of the specification (assuming this is possible), and test purposes or coverage criteria extracted from this execution tree. This is implemented in the Agatha tool [23]. Other approaches, such as [25] or [21], respectively implemented in Gatel and BZ-TT, rely on constraint solving techniques to compute paths to a goal. These approaches are limited to deterministic systems, and consider finite unfoldings of systems by limiting the search depth. In [22] the authors use selection hypotheses combined with operation unfolding for algebraic data types and predicate resolution to produce test cases from Lotos specifications. Compared to our approach, these techniques based on constraint solving may produce more precise test cases. However, constraint solving does not allow to cope with loops as is possible with abstract interpretation, but have to limit the unfolding

to a bounded depth. Nevertheless, these should not be considered as opposite methods, but as complementary ones. Test selection with test purposes using approximate analyses can be seen as a front-end used to select an abstract test case, where information on non-conformance is preserved. Then constraint solving techniques can be used to search for instantiated test cases, by limiting the unfolding of remaining loops.

Some approaches are mostly based on abstraction. In the context of extended LTS, [27] is a pioneering work, an initial attempt of the one presented here. But test selection was based on a very basic abstraction. In [18], a short version of the work presented here was introduced, with emphasis on the relation between test case accuracy and the precision of the approximation. In [28], the approach is combined with verification. Observers of suspension traces are used to describe negation of safety properties, and model-checked on the specification. Even if this does not succeed, test cases are selected from the specification according to these observers, using the approach described here. Selected test cases can then both detect non-conformance and violation of the safety property by the implementation, but also violation of the safety property by the specification if the model-checking phase was not complete.

In most other approaches funded on abstractions, one tries to generate instantiated test cases, i.e., with fixed values for input and output parameters, that exercise particular executions of the system. This is the case for [7], where the idea is first to build an abstraction of a μCRL specification, to generate abstract test cases by reusing the TGV enumerative technique on this abstraction, and then to concretize these test cases on the concrete specification using constraint solving techniques. In the context of white-box testing, Ball [1] uses a combination of predicate abstraction, reachability analysis and symbolic execution.

6 Conclusion and Perspectives

There is still very little research on model-based test generation which is able to cope with models containing both control and data without enumerating data values.

In the present paper, an approach to the off-line selection of test cases from specification models with control and data (ioSTS) and test purposes specified in the same model has been presented. The main advantage of this test generation technique is to avoid the state explosion problem due to the enumeration of data values. Test selection reduces to syntactical operations on these models and relies on an over-approximate analysis of the co-reachable states to a target location. Test cases are generated in the form of ioSTS, thus representing non-instantiated test programs. During execution of test cases on the implementation, constraint solving is used to choose output data values. For simplicity, the theory exposed in this paper is restricted to deterministic specifications. However, non-determnistic specifications can be taken into account with some restrictions [19].

Among the perspectives of this work, more powerful models of systems with features such as time, recursion and concurrency should be considered. For test

generation, one problem to address in these models is partial observability, which, as for ioSTS, entails the identification of determinizable sub-classes corresponding to applications.

Some ideas of these technique can also be used in other contexts, in particular for structural white box testing where test cases are generated from the source code of the system. One of the main problems of these techniques which is to avoid infeasible paths, could be partly solved by techniques similar to those presented here.

Other challenges are the combination of these techniques with coverage-based test selection. Some attempts have been made to define some test coverage criteria by observers [4], which are very similar to test purposes. A combination with our techniques could be beneficial. Another direction should be to use the dynamic partitioning facility (provided by the tool Nbac used by STG) as an aid for test selection with respect to coverage criteria having a deeper semantic meaning.

Acknowledgment

We wish to thank the Organizing Committee of FMCO'06 for this invitation, the reviewers for their useful remarks, as well as all our colleagues for their participation in the work described in this paper.

References

1. Ball, T.: A theory of predicate-complete test coverage and generation. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2004. LNCS, vol. 3657, pp. 2–5. Springer, Heidelberg (2005)
2. Belinfante, A., Feenstra, J., de Vries, R.G., Tretmans, J., Goga, N., Feijs, L., Mauw, S., Heerink, L.: Formal test automation: A simple experiment. In: 12th Int. Workshop on Testing of Communicating Systems, Kluwer Academic Publishers, Dordrecht (1999)
3. Benjamin, M., Geist, D., Hartman, A., Mas, G., Smeets, R., Wolfsthal, Y.: A study in coverage-driven test generation. In: Proceedings of the 36th ACM/IEEE Conference on Design Automation (DAC'99) (1999)
4. Blom, J., Hessel, A., Jonsson, B., Pettersson, P.: Specifying and generating test cases using observer automata. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 137–152. Springer, Heidelberg (2005)
5. Boyer, R.S., Elspas, B., Levitt, K.N.: SELECT: a formal system for testing and debugging programs by symbolic execution. In: Proceedings of the International Conference on Reliable Software, pp. 234–245. ACM Press, New York (1975)
6. Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems. LNCS, vol. 3472. Springer, Heidelberg (2005)
7. Calamé, J.R., Ioustinova, N., van de Pol, J., Sidorova, N.: Data abstraction and constraint solving for conformance testing. In: Proc. of 12th Asia-Pacific Software Engineering Conference (APSEC'05), Taipei, Taiwan, pp. 541–548 (2005)
8. Clarke, D., Jéron, T., Rusu, V., Zinovieva, E.: STG: a symbolic test generation tool. In: Katoen, J.-P., Stevens, P. (eds.) ETAPS 2002 and TACAS 2002. LNCS, vol. 2280, pp. 470–475. Springer, Heidelberg (2002)

9. Cousot, P., Cousot, R.: Abstract intrepretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4th ACM Symposium on Principles of Programming Languages (POPL'77), Los Angeles, CA, pp. 238–252 (1977)
10. Frantzen, L., Tretmans, J., Willemse, T.: Test generation based on symbolic specifications. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, Springer, Heidelberg (2005)
11. Gaston, C., Le Gall, P., Rapin, N., Touil, A.: Symbolic execution techniques for test purpose definition. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, Springer, Heidelberg (2006)
12. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation, pp. 213–223. ACM Press, New York (2005)
13. Gunter, E., Peled, D.: Model checking, testing and verification working together. Formal Aspects of Computing 17(2), 201–221 (2005)
14. Howden, W.E.: Theoretical and empirical studies of program testing. In: Proceedings of the 3rd international conference on Software engineering (ICSE '78), pp. 305–311. IEEE Press, Piscataway, NJ (1978)
15. ISO/IEC 9646: Conformance Testing Methodology and Framework (1992)
16. Jard, C., Jéron, T.: TGV: theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. Software Tools for Technology Transfer (STTT) (octobre 6, 2004)
17. Jeannet, B.: Dynamic partitioning in linear relation analysis. Formal Methods in System Design 23(1), 5–37 (2003)
18. Jeannet, B., Jéron, T., Rusu, V., Zinovieva, E.: Symbolic test selection based on approximate analysis. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, Springer, Heidelberg (2005)
19. Jéron, T., Marchand, H., Rusu, V.: Symbolic determinisation of extended automata. In: 4th IFIP International Conference on Theoretical Computer Science, 2006, Santiago, Chile. SSBM (Springer Science and Business Media) (August 2006)
20. Lee, D., Yannakakis, M.: Principles and Methods of Testing Finite State Machines - A Survey. In: Proceedings of the IEEE, vol. 84(8), IEEE Computer Society Press, Los Alamitos (1996)
21. Legeard, B., Peureux, F., Utting, M.: Automated boundary testing from Z and B. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, Springer, Heidelberg (2002)
22. Lestiennes, G., Gaudel, M.-C.: Testing processes from formal specifications with inputs, outputs and data types. In: 13th International Symposium on Software Reliability Engineering (ISSRE'02), Annapolis, Maryland, IEEE Computer Society Press, Los Alamitos (2002)
23. Lugato, D., Bigot, C., Valot, Y.: Validation and automatic test generation on uml models: the AGATHA approach. Electronics Notes in Theoretical Computer Science 66(2) (2002)
24. Lynch, N., Tuttle, M.: Introduction to IO automata. CWI Quarterly 3(2) (1999)
25. Marre, B., Arnould, A.: Test sequences generation from LUSTRE descriptions: GATEL. In: 15th IEEE International Conference on Automated Software Engineering (ASE'00), p. 229. IEEE Computer Society, Los Alamitos, CA (2000)
26. Petrenko, A., Boroday, S., Groz, R.: Conforming configurations in EFSM testing. IEEE Transactions on Software Engineering 30(1) (2004)

27. Rusu, V., du Bousquet, L., Jéron, T.: An approach to symbolic test generation. In: Grieskamp, W., Santen, T., Stoddart, B. (eds.) IFM 2000. LNCS, vol. 1945, pp. 338–357. Springer, Heidelberg (2000)
28. Rusu, V., Marchand, H., Jéron, T.: Automatic verification and conformance testing for validating safety properties of reactive systems. In: Fitzgerald, J.A., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, Springer, Heidelberg (2005)
29. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software—Concepts and Tools 17(3), 103–120 (1996)
30. Tretmans, J.: Model-based testing with transition systems. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4111, Springer, Heidelberg (2006)